Rendering

Charles Hansen

University of Utah,
Salt Lake City, UT, USA, 84112.

E. Wes Bethel

Computational Research Division
Lawrence Berkeley National Laboratory,
Berkeley, California, USA, 94720.

Thiago Ize

University of Utah,
Salt Lake City, UT, USA, 84112.

Carson Brownlee

University of Utah,
Salt Lake City, UT, USA, 84112.

October 2012

Acknowledgment

This work was supported by the Director, Office of Science, Office and Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Some of the research in this work used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Legal Disclaimer

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor The Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or The Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or The Regents of the University of California.

Preface

The material in this technical report is a chapter from the book entitled *High Performance Visualization—Enabling Extreme Scale Scientific Insight* [2], published by Taylor & Francis, and part of the CRC Computational Science series.

Contents

1 Introduction

The process of creating an image from the data, as the last step before display, in the visualization pipeline is called *rendering*; this is a key aspect of all high performance visualizations. The rendering process takes data resulting from visualization algorithms and generates an image for display. This figure highlights that data is typically filtered, transformed, subsetted and then mapped to renderable geometry before rendering takes place. There are two forms of rendering typically used in visualization, based on the underlying mapped data: rendering of geometry generated through visualization mapping algorithms and direct rendering from the data. For *geometric rendering*, typical high performance visualization packages use the OpenGL library, which converts the geometry generated by visualization mapping algorithms to colored pixels through rasterization [37]. Another method for generating images from geometry is ray tracing, where rays from the viewpoint through the image pixels, are intersected with geometry to form the colored pixels [36]. *Direct rendering* does not require an intermediate mapping of data to geometry, but rather, it directly generates the colored pixels through a mapping that involves sampling the data and for each sample, mapping the data to renderable quantities (color and transparency). Direct volume rendering is a common technique for rendering scalar fields directly into images.

The rest of this report will discuss a rendering taxonomy that is widely used in high performance visualization systems. Geometric rendering is then presented with examples of both rasterization and ray tracing solutions. Direct volume rendering is introduced. And finally, an example of a geometric rendering system is discussed, using ray tracing on a commodity cluster.

2 Rendering Taxonomy

In 1994, Molnar et al. described a taxonomy of parallel rasterization graphics hardware that has become the basis for most parallel implementations—both hardware and software based—and is widely used in high performance visualization systems [26]. While the taxonomy describes different forms of graphics hardware, the generalization of the taxonomy provides the basis for most software-based and GPU-based parallel rendering systems for both rasterization rendering and direct volume rendering.

The taxonomy describes three methods for the parallelization of graphics hardware performing rasterization, based on when the assignment to the image space (sometimes called screen-space) takes place. If one considers the rasterization process of consisting of two basic steps, geometry processing and rasterization (the generation of pixels from the geometry), then the sort (the assignment of data, geometry, or pixels), to image space can occur at three points.

As shown in the left of Figure 1, geometric primitives can be assigned to screen-space before geometry processing takes place. This is called sort-first. Since geometry processing involves the transformation of the geometry to image space, this requires *a priori* information about the geometry and the mapping to image space. Such *a priori* information can be obtained from previously rendered frames, heuristics, or by simply replicating the data among all parallel processes. The image space is divided into multiple tiles, each of which can be processed in parallel. Since the geometry is assigned to the appropriate image space tile, geometry processing, followed by rasterization, generates the final colored pixels of the image. This process can be conducted in parallel by each of the processors responsible for a particular image tile. The advantage of sort-first methods is that image compositing is not required, since subregions of the final image are defined uniquely. The disadvantage is that, without data replication or heuristics, the assignment of data to the appropriate image space subregion is difficult. Sort-first methods typically leverage the frame-to-frame coherency, which is often readily available in interactive applications.

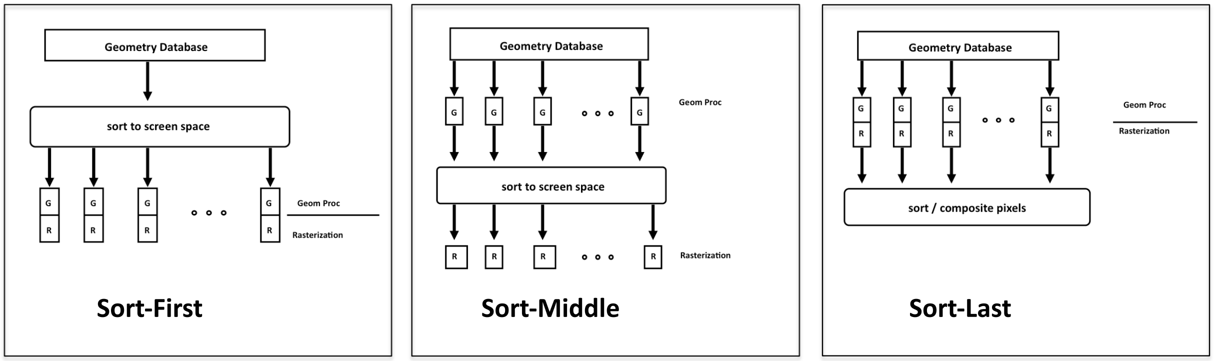


Figure 1: The rendering taxonomy based on Molnar's description. Left most is sort-first, the center is sort-middle, while the right most is sort-last.

Sort-middle architectures perform the sort to image space, after geometry processing, but before rasterization. As in sort-first, the image is divided into tiles and each rasterization processor is responsible for a particular image tile. Geometry is processed in parallel by the multiple geometry processors that obtain geometry in some manner, such as round-robin distribution. One of the steps of geometry processing is the transformation of the geometry to image space. The image space geometry is sorted and sent to the appropriate rasterization processor responsible for the image space partition covered by the processed geometry. While common in graphics hardware, high performance visualization systems typically use either sort-last or sort-first methodology for parallel rendering.

The right-most image in Figure 1 shows the sort-last technique. In this method, geometry is distributed to geometry processors, in some manner, such as round-robin distribution. Each processor in parallel transforms the geometry to image space and then, rasterizes the geometry to a local image. The rasterizer generates the image pixels. Note that depending on the geometry, the pixels may cover an entire image, or more typically, a small portion of the image. After all the geometry has been processed and rasterized, the resulting partial images are combined, through compositing. The advantage of the sort-last method is that renderable entities, geometry, or data can be partitioned among the parallel processors and the final image is constructed through parallel compositing. The disadvantage is that the compositing step can dominate the image generation time. Sort-last methods are the most common found in high performance visualization and are part of the VisIt and ParaView distributions.

3 Rendering Geometry

As described above, one method for visualization involves mapping the data to geometry, such as an isosurface or an intermediate representation, such as spheres, and then rendering the geometry into an image for display. The send-geometry method generates such geometry for rendering on a client. The rendering can be performed either serially, or in parallel and in either software or more typically through GPU hardware. The most common rasterization library for performing this task is the OpenGL library. OpenGL is an industry standard graphics API, with render implementations supporting GPU, or hardware-accelerated rendering. There is a long history of research in parallel and distributed geometry rasterization, focusing on both early massively parallel processors (MPP) and clusters of PCs.

Crockett and Orloff [10] describe a sort-middle parallel rendering system for message-passing machines. They developed a formal model to measure the performance and they were one of the first to use MPPs for parallel rendering. In comparison, Ortega et al. [31] introduce a data-parallel geometry rendering system using sort-last compositing. This implementation allowed integrated geometry extraction and rendering on the CM5 and was targeted for applications, which required extremely fast rendering for extremely large sets of polygons. The rendering toolkit enables the display of 3D shaded polygons directly from a parallel machine, avoiding the transmission of huge amounts of geometry data to a post-processing rendering system over the then slow networks. Krogh et al. [21] present a parallel rendering system for molecular dynamics applications, where the massive particle system is represented as spheres. The spheres are rendered with depth-enhanced sprites (screen aligned textures) and the results are composited with sort-last compositing. This system avoids rasterization by the use of sprites.

Correa et al. [8] present a technique for the out-of-core rendering of large models on cluster-based tiled displays, using sort-first. In their study, a hierarchical model is generated in a preprocess and at runtime, each node renders an image-tile in a multi-threaded manner, overlapping rendering, visibility computation, and disk operations. By using 16 nodes, they were able to match the performance of the, at the time, current high-end graphics workstations. Samanta et al. [35] introduce k-way replication for sort-first rendering where geometry is only replicated on k out of n nodes with k being much smaller than n. This small replication factor avoids the full data replication, typically required for sort-first rendering. It also supports rendering large-scale geometry, where the geometry can be larger than the memory capacity of an individual node. It simultaneously reduces communication overheads by leveraging frame-to-frame coherence, but allowing a dynamic, view-dependent partitioning of the data. The study found that parallel rendering efficiencies, achieved with small replication factors were, similar to the ones measured with full replication. VTK and VisIt use the Mesa OpenGL library, which implements the OpenGL API in software, to enable parallel rendering through sort-last compositing. Rendering, with either VisIt or Paraview, scales well on GPU clusters using hardware rendering and sort-last compositing, but they are bounded by the composite time for the image display. When using Mesa, the rendering is not scalable or interactive, due to the rasterization speed; though, replacing rasterization with ray tracing can improve the scalability of software rendering [5].

An alternative to rasterization for high performance visualization is to render images using ray tracing. Ray tracing refers to tracing rays from the viewpoint, through pixels, and intersecting the rays with the geometry to be rendered. There have been several parallel ray tracing implementations for high performance visualization. Parker et al. [32] implement a shared-memory ray-tracer, RTRT, which performs interactive isosurface generation and rendering on large shared-memory computers, such as the SGI Origin. This proves to be effective for large data sets, as the data and geometry are accessible by all processors. Bigler et al. [3] demonstrate the effectiveness of this system on large-scale scientific data. They extend RTRT to use various methods in order to visualize the particle data, from simulations using the material point method, as spheres. They also describe two methods for augmenting the visualization using silhouette edges and advanced illumination, such as ambient occlusion. In Section 5, there are details about how to accelerate ray tracing for large amounts of geometry on distributed-memory platforms, like commodity clusters.

4 Volume Rendering

Direct volume rendering methods generate images of a 3D volumetric data set, without explicitly extracting geometry from the data [13, 22]. Conceptually, volume rendering proceeds by casting rays from the viewpoint through pixel centers as shown in Figure 2. These rays are sampled where they intersect the data volume. Volume rendering uses an optical model to map sampled data values to optical properties, such as color and opacity [25]. During rendering, optical properties are accumulated along each viewing ray to form an image of the data, as shown in Figure 3. Although the data set is interpreted as a continuous function in space, for practical purposes it is represented as a discrete 3D scalar field. The samples typically are trilinearly interpolated data values from the 3D scalar field. On a GPU, the trilinear interpolation is performed in hardware and, therefore, is quite fast. Samples along a ray can be determined analytically, as in ray marching [36], or can be generated through proxy geometry on a GPU [15].

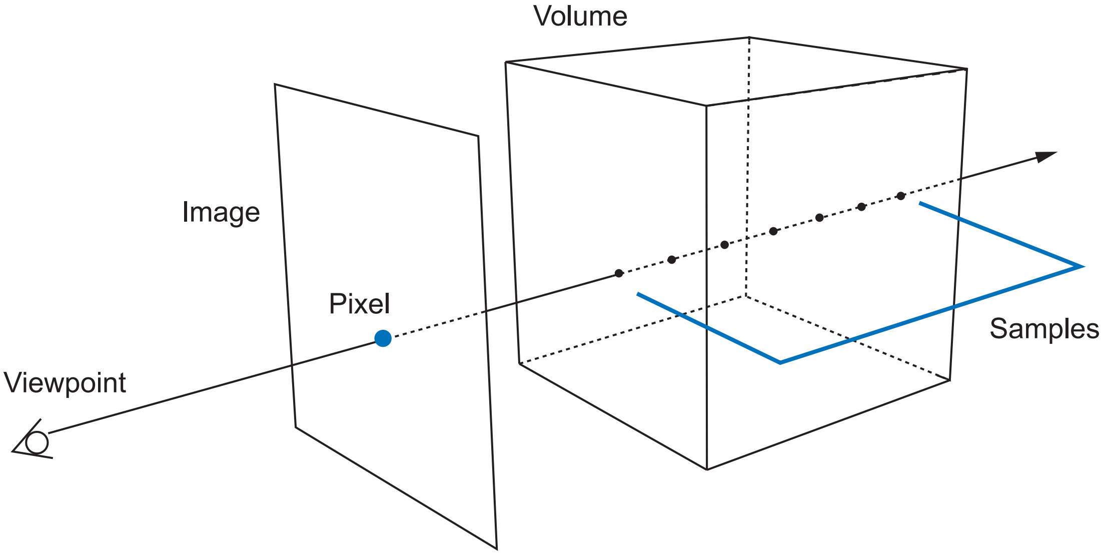


Figure 2: Ray casting for volume rendering.

In the quest towards interactivity and towards addressing the challenges posed by growing data size and complexity, there has been a great deal of work over the years on parallel volume visualization (see Kaufman and Mueller [18] for an overview). Volume rendering is easily parallelized, though care must be taken to achieve effective results. Typically, the 3D scalar field is partitioned into sub-volumes, which are assigned to the parallel nodes. Volume rendering can take place asynchronously, within the parallel nodes. Also, volume rendering synchronizes compositing between the parallel nodes, forming the final image.

The TREX system [20] is a parallel volume rendering application for shared-memory platforms that uses object-parallel data domain decomposition and texture-based, hardware-accelerated rendering, followed by a parallel, software-based composition phase with image space partitioning. TREX can map different portions of its pipeline to different system components on the SGI Origin; those mappings are intended to achieve optimal performance at each algorithmic stage and to achieve minimal inter-stage communication costs.

Muraki et al. [29] describe a custom hardware-based compositing system linking commodity GPUs used for volume rendering. They implemented a cluster composed of two 8-node systems that are linked by multiple networks. While the CPUs are used for volume processing, the GPUs are used for volume rendering and, custom compositing hardware forms the final image in a sort-last manner. Müller et al. [28] implement a ray casting volume renderer with object-order partitioning

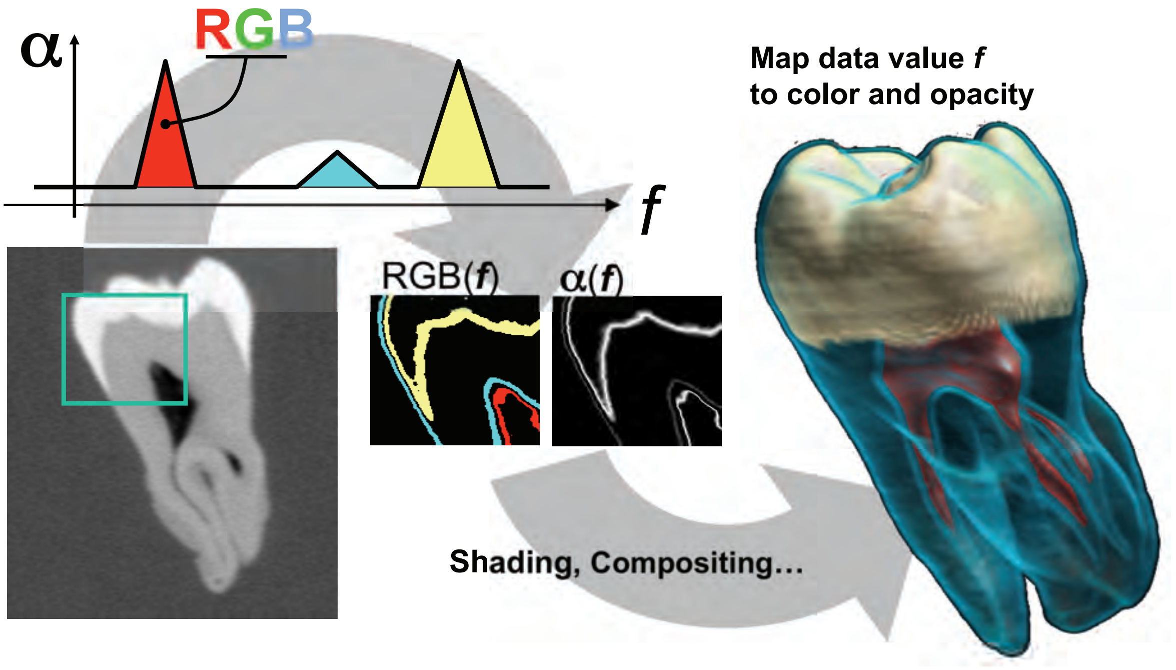


Figure 3: Volume rendering overview. The volume is sampled, samples are assigned optical properties (RGB, α). The different properties are shaded and composited to form the final image. Image courtesy of Charles Hansen (University of Utah).

on a small 8-node GPU cluster, using programmable shaders. This system sustains frame rates in the single digits for data sets, as large as 1260^3 with an image size of 1024×768 pixels. More recently, Moloney et al. [27] implement a GLSL, texture-based volume renderer that runs on a 32-node GPU system. This work takes advantage of the sort-first architecture to accelerate certain types of rendering, like occlusion culling.

Peterka et al. [33] discuss end-to-end performance of parallel volume rendering on an IBM Blue Gene distributed-memory parallel architecture. They explore the system performance in terms of rendering, compositing and disk I/O. The system employs sort-last compositing using direct-send compositing. Their system was useful for very large data sets that could not be accommodated by GPU clusters and could produce frame times on the order of a few seconds for such data.

Yu et al. [41] describe an *in situ* parallel volume rendering implementation. The data partitioning is used directly from the domain decomposition for the corresponding simulation. Rendering combines both rasterization for geometry, representing particles, and volume rendering of the associated scalar field. Sort-last compositing, using 2-3 swap, was employed to form the final image. Their system tightly linked the visualization with the simulation and scaled to 15,360 cores.

Childs et al. [7] present a parallel volume rendering scheme for massive data sets (with one hundred million unstructured elements and a 3000^3 rectilinear data set). Their approach parallelizes over both input data elements and output pixels, which is demonstrated to scale well on up to 400 processors.

A hybrid approach to parallel volume rendering, which makes use of a design pattern common in many parallel volume rendering applications, uses a mixture of both object- and pixel-level parallelism [1, 23, 24, 38] The design employs an object-order partitioning that distributes source data blocks to processors where they are then rendered using ray casting [13, 22, 34, 39]. Within a processor, an image space decomposition is used, similar to Nieh and Levoy [30], to allow multiple rendering threads to cooperatively generate partial images that are later combined via compositing into a final image [13, 22, 39]. This hybrid design approach, which uses a blend of object- and pixel-level parallelism, has proven successful in achieving scalability and tackling large data sizes.

5 Real-Time Ray Tracer for Visualization on a Cluster

While the core ray tracing algorithm might be embarrassingly parallel, scaling a ray tracer to render millions of pixels at real-time frame rates on a cluster remains challenging. It is even more so if the individual nodes do not have enough memory to contain all the data and associated ray tracing acceleration structures (such as octrees or bounding volume hierarchies). The ability to render high-resolution images at interactive or real-time rates is important when visualizing data sets that contain information at the subpixel level. Since most commodity monitors are now capable of displaying at least a two megapixel HD resolution of 1920 × 1080, and higher end models can display up to twice that many pixels, it is important to make use of all those pixels when visualizing a data set. Current distributed ray tracing systems are not able to approach interactive rates for such pixel counts, regardless of how many compute nodes are used. Either lower resolutions are required for real-time rates, or less fluid frame rates are used in order to scale to larger image sizes. This section describes a distributed ray tracing system developed by Ize et al. that can scale, on an InfiniBand cluster, to real-time rates of slightly over 100fps, at full HD resolution, or 50–60fps at 4 megapixels with massive polygonal models [16]. The system uses sort-first parallelism, where the image is divided into tiles, but rather than pre-sort and assign geometry to nodes, the geometry is dynamically cached to the appropriate node taking advantage of frame-to-frame coherence.

Such a system can also handle massive out-of-core models that cannot reside inside the physical memory of any individual compute node by implementing a read-only distributed shared memory ray tracer [9] which is essentially a distributed cache (DC). The DC supports the use of any desired ray tracing shading models, such as shadows, transparent surfaces, ambient occlusion and even full path tracing. More advanced shading models than simple ray casting or rasterization allow for more productive and useful visualizations [14].

Ize et al. implemented their distributed ray tracer on the Manta Interactive Ray Tracer, a state-of-the-art ray tracer capable of scaling at real-time frame rates to hundreds of cores on a shared-memory machine [4]. They faced the challenge of ensuring that the distributed Manta implementation was able to scale to many nodes while also ensuring real-time frame rates.

The two main challenges of a real-time distributed ray tracing system are load balancing and ensuring the display is not a bottleneck. The real-time distributed ray tracer uses a master-slave configuration where a single process, the display, receives pixel results from render processes running on the other nodes. Another process, the load balancer, handles assigning tasks to the individual render nodes.

MPI does not guarantee fairness amongst threads in a process and the MPI library, in their implementation, forces all threads to go through the same critical section. Because of this, the display and load balancer were run as separate processes. Since they do not communicate with each other, nor do they share any significant state, this partitioning can be done without penalty or code complexity. Furthermore, since the load balancer has minimal communication, instead of running on a dedicated node, it can run alongside the display process without noticeably impacting overall performance.

5.1 Load Balancing

Manta uses a shared-memory dynamic load balancing work queue, where each thread is statically given a predetermined large tile of work to consume and then, when it needs more work, it progressively requests smaller tiles until there is no more work left. Ize et al. extended Manta's shared-memory load balancer to distributed memory, using a master dynamic load balancer with a work queue comprised of large tiles, which are given to each node (the first assignment is done statically and is always the same) and then each node has its own work queue where it distributes sub-tiles to each render thread. Inside each node, the standard Manta shared-memory load balancer distributes work amongst the threads. Each thread starts with a few statically assigned ray packets to render and then takes more ray packets from the node's shared work queue until no more work is available, at which point that thread requests another tile of work from the master load balancer for the entire node to consume. This approach effectively provides a two-level load balancer ensures that work is balanced both at the node level and at the thread level. Since the top level load balancer only needs to keep the work queues of the nodes full, instead of the queues for each individual thread, communication is kept low on the top level load balancer, which allows the system to scale to many nodes and cores.

5.2 Display Process

One process, the display, is dedicated to receiving pixels from the render nodes and placing those pixels into the final image. The display process shares a dedicated node with the load balancer process, which only uses a single thread and has infrequent communication. The display has one thread, which only receives the pixels from the render nodes into a buffer. The other threads in the display then take those pixels and copy them into the relevant parts of the final image. At first, Ize et al. used only a single thread to do both the receiving and copying to the final image, but they found that the maximum frame rates were significantly lower than what the InfiniBand network should be capable of. Surprisingly, it turned out that merely copying data into the local memory was introducing a bottleneck; and, for this reason, they employed several cores to do the copying.

Since InfiniBand packets are normally 2KB, any messages smaller than this 2KB will still consume 2KB of the network bandwidth. Therefore, it is necessary to send a full packet of pixels to maximize frame rate. If, for instance, one sends only a single seven-byte pixel at a time, it would actually require sending an effective $2048 \times 1920 \times 1080 = 3.96$GB of data, taking about two seconds over the high speed network, which is clearly too slow. Since ray packets contain 64 pixels, combining 13 ray packets into a single message offers the best performance, since it uses close to three full InfiniBand packets.

5.3 Distributed Cache Ray Tracing

The distributed cache infrastructure is similar to that of DeMarle et al. [12]. Data is distributed among nodes in an interleaved pattern. A templated direct-mapped cache is used for data that does not exist locally. Data is accessed at a block granularity of almost 8KB, with a little bit of space left for packet/MPI overhead, which results in each block containing 254 32-byte Bounding Volume Hierarchy (BVH) nodes or 226 36-byte triangles. Since multiple threads can share the cache, each cache line is controlled by a mutex so that multiple threads can simultaneously read from a cache line. In order to replace the data in a cache line, a thread must have exclusive access to that cache line so that when it replaces the cache element, it does not modify data that is being used by other threads. Remote reads are performed using a passive `MPI_Get` operation, which should, in turn, use an InfiniBand RDMA read to efficiently read the memory from the target node without any involvement of the target CPU. This approach allows for very fast remote reads, which do not impact performance on the target node and that scale to many threads and MPI processes [17].

DeMarle et al. assign block k to a node number $k \bmod numNodes$, so that blocks are interleaved across a distributed memory. If a node owns block k, it will then place it in location $k/numNodes$ of its resident memory array [11]; Ize et al. follow this convention. However, if a node does not own block k, the node places a copy of the block in its $k \bmod cacheSize$ cache line. Ize et al. found this to be an inefficient mapping, because it does not make full utilization of the cache, as it does not factor in that some of the data might already reside in the node's resident memory. For instance, suppose, there are 2 nodes and the cache size is also 2, then node 0 will never be able to make use of cache line 0. When $k \bmod 2 = 0$, then the owner of the data is $k/2 = 0$, which means that node 0 already has that data in its resident memory. A more efficient mapping that avoids the double counting is:

$$\left(k - \left\lfloor \frac{k + (numNodes - myRank)}{numNodes} \right\rfloor\right) \bmod cachSize.$$

Ray casting the Richtmyer–Meshkov data set (Fig. 4) with 60 nodes and more efficient mapping gives speedups of 1.16×, 1.31×, 1.48×, 1.46×, and 1.31× over the mapping of DeMarle et al. The respective cache sizes are 1/32, 1/8, 1/4, 1/2, and 1/1 of total memory, respectively.

5.3.1 DC BVH

Since the DC manager groups BVH nodes into blocks, Ize et al. reorder the memory locations of the BVH nodes so that the nodes in a block are spatially coherent in memory. Note, they are not reordering the actual BVH tree topology. They accomplish this coherency for a block size of B, BVH nodes by writing the nodes to memory according to a breadth first traversal of the first B nodes, thus creating a subtree that is coherent in memory. They then stop the breadth first traversal and instead recursively repeat that process for each of the B leaves of the newly formed subtree. If the subtree being created ends up with less than B leaves, it continues to the next subtree without introducing any gaps in the memory layout. It is therefore possible for a block to contain multiple subtrees. However, since the blocks are written according to a blocked depth-first traversal, the subsequent subtree will still often be spatially near the previous subtree.

In order to minimize memory usage, Manta's BVH nodes only contain a single child pointer rather than two, with the other child's memory location being adjacent to the first child. Because of this, the algorithm above recurses on each pair of child leaves, so that the two children stay adjacent in memory, rather than recursing on each of the B leaves.

Thus, the spatially coherent blocks contain mostly complete subtrees of B nodes so that when a block is fetched, one can usually expect to make $\log B$ traversals before a new block must be fetched from the DC manager. For 254 node blocks, this is about eight traversal steps for which the DC-BVH traversal performance should be roughly on par with the regular BVH traversal. Eight traversal steps are ensured by keeping track of the current block and not releasing that block (or re-fetching it) until either the traversal leaves the block or enters a new block. When the traversal enters a new block it must release the previously held block in order to prevent a deadlock condition where one of the following blocks requires the same cache line as the currently held block. Note, all multiple threads can safely share access to a block and thread stalling, while waiting for a block to be released, will only occur if one thread needs to use the cache line for a different block than is currently being held.

Since the root of the tree will be traversed by every thread in all nodes, rather than risk this data being evicted and then having to stall while the data becomes available again, the corresponding spatially coherent block is replicated across all nodes. Replication requires only an extra 8KB of data per node and ensures that those first 8 traversal steps are always fast because they only need to access resident memory.

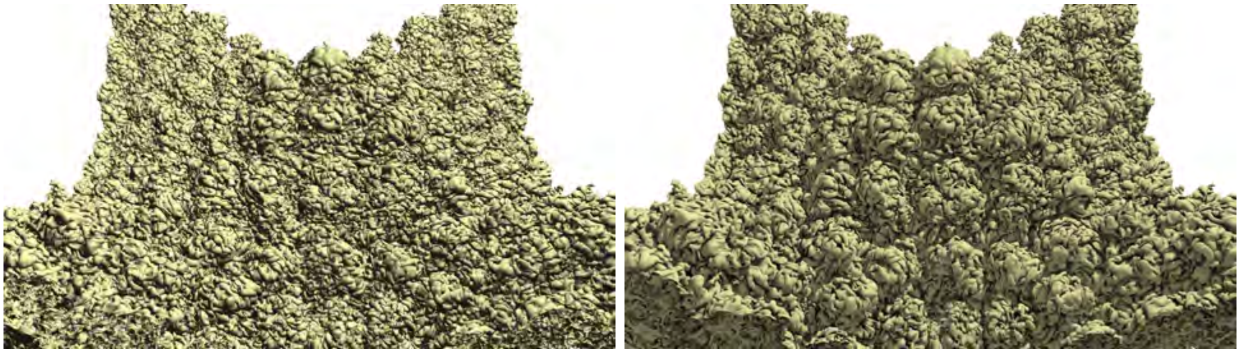


Figure 4: On 60 nodes, the system can ray cast (left image) a 316M triangle isosurface from timestep 273 of a Richtmyer–Meshkov (RM) instability calculation, in HD resolution, at 101fps, with replicated data, 21,326MB of triangles and acceleration structures, and at 16fps if the DC is used to store only 2,666MB per node. Using one shadow ray or 36 ambient occlusion rays per pixel (right image) the system can achieve 4.76fps and 1.90fps, respectively. Image source: Ize et al. [16].

5.3.2 DC Primitives

While sharing vertices using a mesh will often halve the memory requirements, shared vertices do not map well to the DC since shared vertices require fetching a block from the DC manager to find the triangle and then once the vertex indices are known, one to three more fetches for the vertices. Thus, a miss results in between a 2×–4× slowdown for misses. Doubling the storage requirements is comparable in cost to halving the cache size, and halving the cache size empirically introduces less than a 2× performance penalty. Although DeMarle et al. used a mesh structure, Ize et al. did not. While it might appear that one could create a sub-mesh within each block so that less memory is used and only a single block need be fetched, due to a fixed block size, there will be wasted empty space inside each block. Attempting to place variable numbers of triangles in each block, so that empty space is reduced, it will also not work since then, it will not be possible to compute which block key corresponds to which triangle.

Ize et al. reorder the triangles into blocks by performing an in-order traversal of the BVH and outputting triangles into an array as they are encountered. This results in spatially coherent blocks that also match the traversal pattern of the BVH so that, if two leaf nodes share a recent ancestor, they are also spatially coherent and will likely have their primitives residing in the same block.

5.4 Results

Ize et al. use a 64 node cluster where each node contains two 4-core Xeon X5550s running at 2.67GHz, with 24GB of memory, and a 4× DDR InfiniBand interconnect between the nodes. All scenes are rendered at an HD resolution of 1920 × 1080 pixels. The data sets consist of a

316M triangle isosurface, computed from one timestep of a Richtmyer–Meshkov (RM) instability calculation (see Fig. 4) and the 259M triangle Boeing 777 CAD model (see Fig. 5). While the RM data set can be rendered using volume ray casting, this polygonal representation is used as an example of a massive polygonal model, where large parts of the model can be seen from one view. The Boeing data set consists of an almost complete CAD model for the entire aircraft and, unless it were made transparent, has significant occlusion so that regardless of the view, only a fraction of the scene can be viewed at any given time.

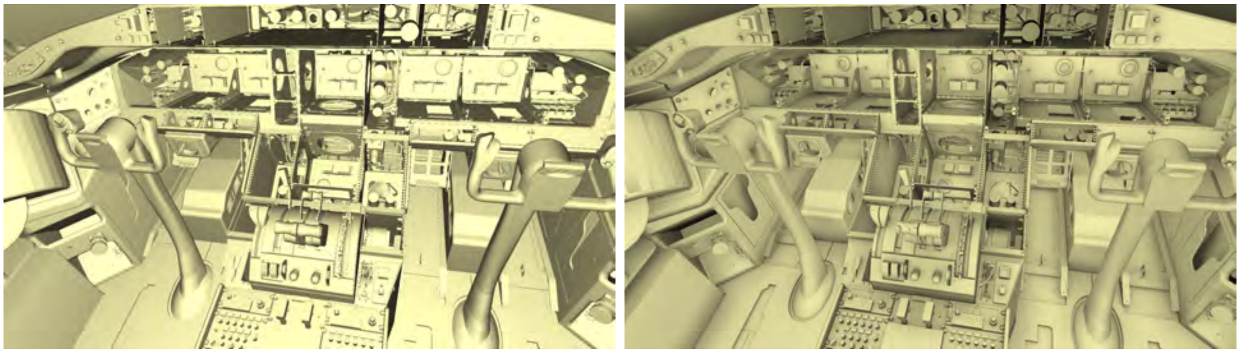


Figure 5: On 60 nodes, the 259M triangle Boeing data set using HD resolution can be ray cast (left image) at 96fps if all 15,637MB of triangle and acceleration structure data are replicated on each node and at 77fps if DC is used to store only 1,955MB of data and cache per node. Using 36 ambient occlusion rays per pixel (right image) the system achieves 1.46fps. Image source: Ize et al. [16].

In the study, scaling was reported where both data sets were rendered using two to sixty nodes, with one node used for display and load balancing, and the remaining nodes used for rendering. Simple ray casting was used for both models and ambient occlusion with 36 samples per shading point for the Boeing data set and a similar ambient occlusion, but with additional hard shadows for the RM data set.

Figure 6 shows how the system scales with increasing numbers of nodes—when ray casting both scenes—using replicated data across each node so that the standard BVH acceleration structure is used without any DC overhead. Then, a cache plus resident set size, $1/N$, of the total memory is used by the data set and acceleration structure. Assuming the nodes have enough memory, data replication allows the system to achieve near linear scaling to real-time rates and then begins to plateau as the rendering rate approaches 100fps. Since 1/1 has a cache large enough to contain all the data, no cache misses ever occur. As the cache is decreased in size, the rendering speed of the Boeing data set is not significantly impacted; indicating that the system was able to keep the working set fully in cache. The RM data set, on the other hand, has a larger working set, since more of it is in the view frustum, which causes cache misses to occur, and noticeably affects performance.

The total amount of memory required in order to keep data in-core depends on the cache size and the resident set size. On a single render node, all the data must be resident. For the RM data set, the resident set is 21GB and the cache is 0GB. Figure 7 shows that with more render nodes the size of the resident set becomes progressively smaller so that the cache size quickly becomes the limiting factor as to how much data the system can handle. Since the resident set decreases as more nodes are added, cache size can similarly be increased so that the node's capabilities are used to the fullest.

5.5 Maximum Frame Rate

Frame rate is limited by the cost of transferring pixels across the network to the display process. The 4×DDR InfiniBand interconnect has a measured bandwidth of 1868MB/s when transferring multiples of 2KB of data according to the system supplied `ib_read_bw` tool. Each pixel sent across consists of a 3 byte RGB color and a 4 byte pixel location. The 4 bytes used for the location is required in order to support rendering modes where pixels in a ray packet could be randomly distributed about the image, for instance, with frameless rendering. The 4 byte location can be removed if the rays within a ray packet form a rectangular tile, in which case it is only necessary to store the coordinates of the rendered rectangle over the entire ray packet instead of 4 bytes per ray. Further improvements might be obtained by compressing the pixels so that even less data needs to be transmitted. The time required by the display process to receive all the pixels from the other nodes is at best $\frac{1920*1080*7B}{1868MB/s} = 7.41$ms for a full HD image, which is an upper limit of 135fps. Since $13 * 64 = 832$ pixels are sent at a time, which occupies 95% of the InfiniBand packet, one would expect the implementation to achieve at best 127fps.

To see how close the system can approach the maximum frame rate by testing the system overhead, an HD image is rendered from a small model with the camera pointing away from the model, producing a blank screen and requiring only a minimal amount of ray tracing. To verify that the load balancer is not significantly competing for resources, the system is tested with the load balancer on its own dedicated node, and then, with the load balancer sharing the node with the display process. The render nodes thus have a limited amount of work to perform and the display process quickly becomes the bottleneck. Figure 8 shows that when using one thread to receive the pixels and another to copy the pixels from the receive buffer to the image, the maximum frame rate is 55fps, no matter how many render threads and nodes being used. Using two threads to copy the pixels to the image results in up to a 1.6× speedup, three copy threads improves performance by up to 2.3×, but additional copy threads offer no additional benefit, demonstrating that copying is the bottleneck until three copy threads are used, after which the receive thread becomes the bottleneck since it is not able to receive the pixels fast enough to keep the copy threads busy. When around 18 threads are being used, be they on a few nodes or many nodes, the system can obtain a frame rate of 127fps. This is exactly the expected maximum of 127fps given by the amount of time it takes to transmit all the pixel data across the InfiniBand interconnect. More render threads result in lower performance due to the MPI implementation not being able to keep up with the large volume of communication. In order to achieve the results required, the MPI implementation must be tuned to use more RDMA buffers and turn off shared receive queues (SRQ); otherwise, the system can still achieve the same maximum frame rate of about 127fps with 17 render cores, but after that point, adding more cores causes performance to drop off quickly, with 384 render cores (48 render nodes) being 2× slower. However, this is a moot point since faster frame rates will offer no tangible benefit.

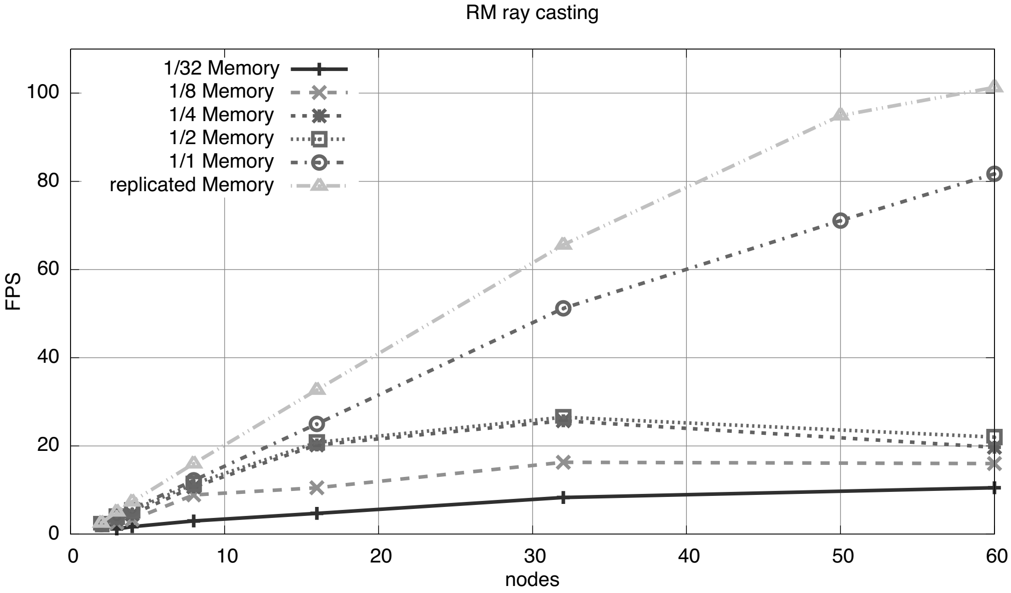
Modern graphics cards can produce four megapixel images at 60fps. As this image size is roughly twice the HD image size, in our system the maximum frame rate would halve to about 60fps. Higher resolutions than 4 megapixels are usually achieved with a display wall consisting of a cluster of nodes driving multiple screens. In this case, the maximum frame rate will be given not by the time to transmit an entire image, but by the time it takes for a single display node to receive its share of the image. Assuming each node renders 4 megapixels, and the load balancing and rendering continue to scale, the frame rate will thus stay at 60fps, regardless of the resolution of the display wall.

Since three copy threads are able to keep up with the receiving thread, and the load balancer process is also running on the same node, there are three unused cores on the tested platform. If data is replicated across the nodes then these three cores can be used for a render process. This render process will also benefit from being able to use the higher-speed shared memory for its MPI communication with the display and load balancer instead of the slower InfiniBand. However, if DC is required, then it will not be possible to run any render processes on the same node since those render processes will be competing with the display and load balancer for scarce network bandwidth and this will much more quickly saturate the network port and result in much lower maximum frame rates.

With modern hardware and software, the described system can ray trace massive models at real-time frame rates on a cluster and even show interactive to real-time rates when rendering distributed geometry using a small cache. The system is one to two orders of magnitude faster than previous cluster ray tracing implementations, which used both slower hardware and algorithms [12, 40], or had equivalent hardware but did not scale to as many nodes or to high frame rates [6]. Compared to compositing approaches, the system can achieve about a 4× improvement in the maximum frame rate for same size non-empty images compared to the state of the art [19] and can also handle advanced shading effects for improved visualization.

6 Conclusion

Parallel rendering methods for generating images from visualizations are an important area of research. In this report, a general framework for parallel rendering was presented and applied to both geometry rendering and volume rendering. In the future, as HPV moves into the exascale regime, parallel rendering methods will likely become more important as *in situ* methods require parallel rendering and the send-image method of parallel display will scale better than the send-geometry method. It is anticipated that GPUs will become integrated into compute nodes, which offer another avenue for parallel rendering in HPV.



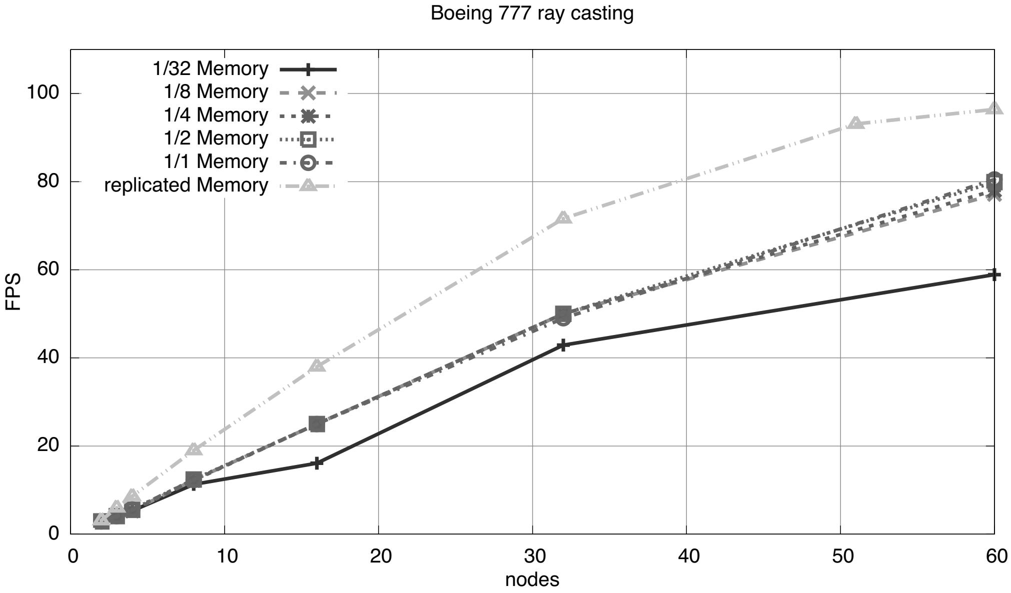


Figure 6: Frame rate when varying cache size and the number of nodes in the ray casted RM and Boeing data sets. The replicated data has almost perfect scaling until the display becomes network bound. Performance scales very well with the Boeing data set and DC, even with a small cache, because the working set is a fraction of the overall model. The scaling with the RM data set and DC performs well until about 20–25fps is reached. Image source: Ize et al. [16].

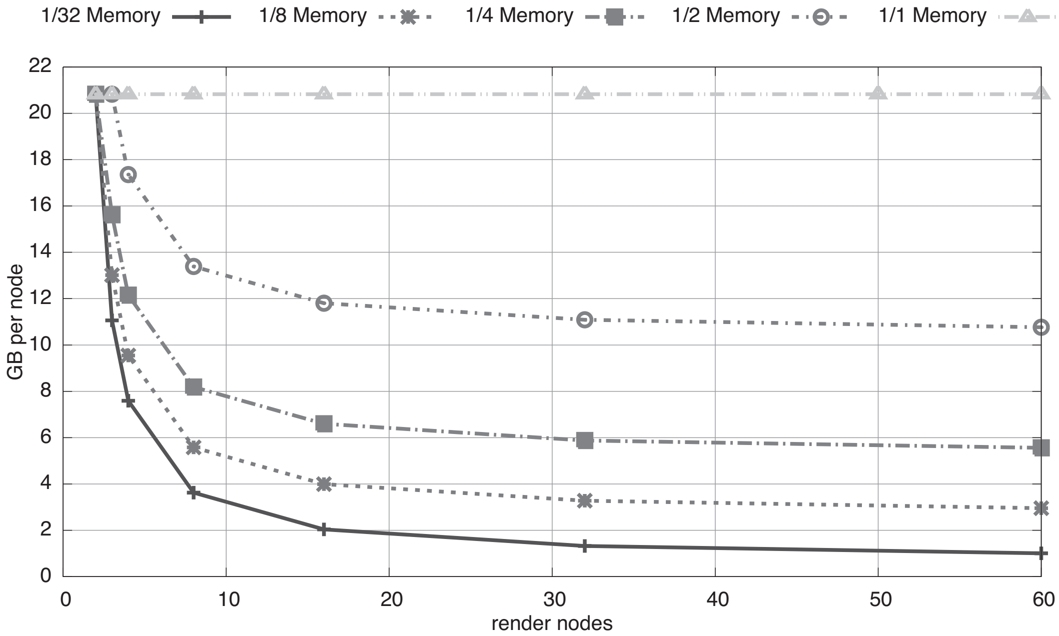


Figure 7: Total memory used per node for the RM data set. Image source: Ize et al. [16].

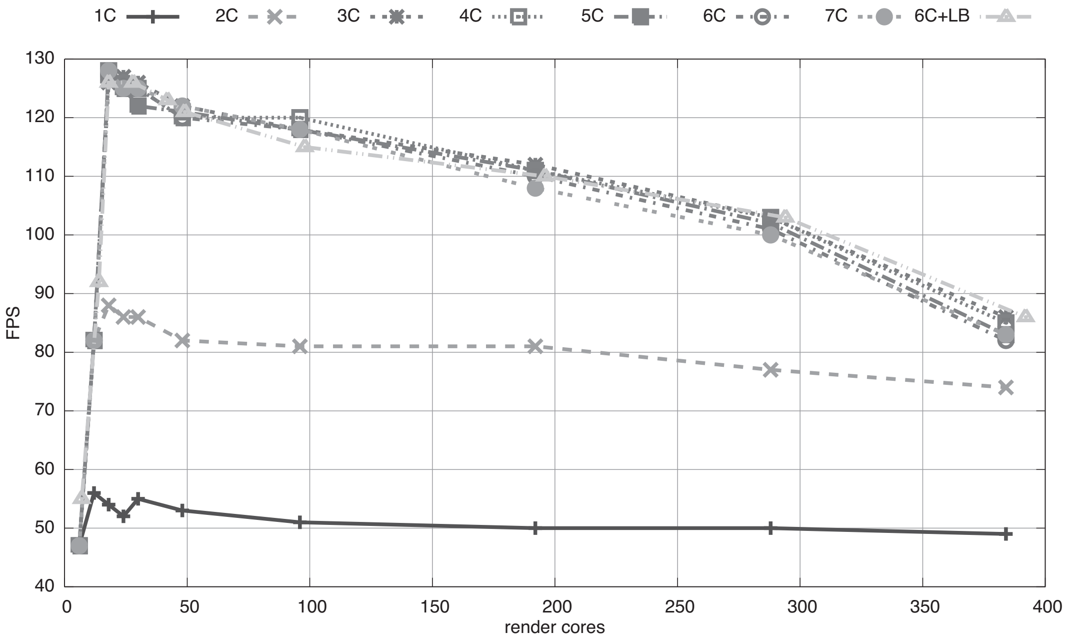


Figure 8: Display process scaling: frame rate using varying numbers of render cores to render a trivial scene when using one copy thread (1C) to seven copy threads (7C), and when using six copy threads with the load balancer process on the same node (6C+LB). Note that performance is not enhanced beyond three copy threads. Image source: Ize et al. [16].

References

[1] C. Bajaj, I. Ihm, G. Joo, and S. Park. Parallel Ray Casting of Visibly Human on Distributed Memory Architectures. In *VisSym '99 Joint EUROGRAPHICS-IEEE TVCG Symposium on Visualization*, pages 269–276, 1999.

[2] E. Wes Bethel, Hank Childs, and Charles Hansen, editors. *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. Chapman & Hall, CRC Computational Science. CRC Press/Francis–Taylor Group, Boca Raton, FL, USA, November 2012. `http://www.crcpress.com/product/isbn/9781439875728`.

[3] James Bigler, James Guilkey, Christiaan Gribble, Charles Hansen, and Steven Parker. A Case Study: Visualizing Material Point Method Data. In *EUROVIS the Eurographics /IEEE VGTC Symposium on Visualization*, pages 299–306. EuroGraphics, 2006.

[4] James Bigler, Abe Stephens, and Steven G. Parker. Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 187–196, 2006.

[5] Carson Brownlee, John Patchett, Li-Ta Lo, David DeMarle, Christopher Mitchell, James Ahrens, and Charles Hansen. A Study of Ray Tracing Large-Scale Scientific Data in Parallel Visualization Applications. In *Proceedings of the Eurographics Workshop on Parallel Graphics and Visualization*, EGPGV '12, pages 51–60. Eurographics Association, 2012.

[6] Brian Budge, Tony Bernardin, Jeff A. Stuart, Shubhabrata Sengupta, Kenneth I. Joy, and John D. Owens. Out-of-Core Data Management for Path Tracing on Hybrid Resources. *Computer Graphics Forum*, 28(2):385–396, 2009.

[7] Hank Childs, Mark A. Duchaineau, and Kwan-Liu Ma. A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 153–162, May 2006.

[8] Wagner T. Corrêa, James T. Klosowski, and Cláudio T. Silva. Out-of-Core Sort-First Parallel Rendering for Cluster-Based Tiled Displays. In *Proceedings of the 4th Eurographics Workshop on Parallel Graphics and Visualization*, EGPGV '02, pages 89–96. Eurographics Association, 2002.

[9] Brian Corrie and Paul Mackerras. Parallel Volume Rendering and Data Coherence. In *Proceedings of the 1993 Symposium on Parallel Rendering*, PRS '93, pages 23–26. ACM, 1993.

[10] Thomas W. Crockett and Tobias Orloff. A MIMD Rendering Algorithm for Distributed Memory Architectures. In *Proceedings of the 1993 Symposium on Parallel Rendering*, PRS '93, pages 35–42. ACM, 1993.

[11] David E. DeMarle. Ice Network Library. `http://www.cs.utah.edu/~demarle/software/`, 2004.

[12] David E. DeMarle, Christiaan Gribble, Solomon Boulos, and Steven Parker. Memory Sharing for Interactive Ray Tracing on Clusters. *Parallel Computing*, 31:221–242, 2005.

[13] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume Rendering. *SIGGRAPH Computer Graphics*, 22(4):65–74, 1988.

[14] Christiaan P. Gribble and Steven G. Parker. Enhancing Interactive Particle Visualization with Advanced Shading Models. In *Proceedings of the 3rd Symposium on Applied Perception in Graphics and Visualization*, pages 111–118, 2006.

[15] Milan Ikits, Joe Kniss, Aaron Lefohn, and Charles Hansen. *Chapter 39, Volume Rendering Techniques*, pages 667–692. Addison Wesley, 2004.

[16] Thiago Ize, Carson Brownlee, and Charles D. Hansen. Real-Time Ray Tracer for Visualizing Massive Models on a Cluster. In *Proceedings of the 2011 Eurographics Symposium on Parallel Graphics and Visualization*, pages 61–69, 2011.

[17] W. Jiang, J. Liu, H.W. Jin, D.K. Panda, D. Buntinas, R. Thakur, and W.D. Gropp. Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science*, 2131:450–457, 2004.

[18] Arie Kaufman and Klaus Mueller. Overview of Volume Rendering. In Charles D. Hansen and Christopher R. Johnson, editors, *The Visualization Handbook*, pages 127–174. Elsevier, 2005.

[19] W. Kendall, T. Peterka, J. Huang, H.W. Shen, and R. Ross. Accelerating and Benchmarking Radix-K Image Compositing at Large Scale. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pages 101–110, 2010.

[20] Joe Kniss, Patrick McCormick, Allen McPherson, James Ahrens, Jamie Painter, Alan Keahey, and Charles Hansen. Interactive Texture-Based Volume Rendering for Large Data Sets. *IEEE Computer Graphics and Applications*, 21(4), July/August 2001.

[21] Michael Krogh, James Painter, and Charles Hansen. Parallel Sphere Rendering. *Parallel Computing*, 23(7):961–974, July 1997.

[22] Marc Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[23] Kwan-Liu Ma. Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed-Memory Architectures. In *PRS '95: Proceedings of the IEEE Symposium on Parallel Rendering*, pages 23–30. ACM, 1995.

[24] Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering. In *Proceedings of the 1993 Parallel Rendering Symposium*, pages 15–22. ACM Press, October 1993.

[25] Nelson Max. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[26] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14:23–32, 1994.

[27] Brendan Moloney, Marco Ament, Daniel Weiskopf, and Torsten Moller. Sort-First Parallel Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1164–1177, 2011.

[28] C. Müller, M. Strengert, and T. Ertl. Optimized Volume Raycasting for Graphics-Hardware-Based Cluster Systems. In *Proceedings of Eurographics Parallel Graphics and Visualization*, pages 59–66, 2006.

[29] Shigeru Muraki, Masato Ogata, Eric Lum, Xuezhen Liu, and Kwan-Liu Ma. VG Cluster: A Low-Cost Solution for Large-Scale Volume Visualization. In *Proceedings of NICOGRAPH International Conference*, pages 31–34, May 2002.

[30] Jason Nieh and Marc Levoy. Volume Rendering on Scalable Shared-Memory MIMD Architectures. In *Proceedings of the 1992 Workshop on Volume Visualization*, pages 17–24. ACM SIGGRAPH, October 1992.

[31] Frank Ortega, Charles D. Hansen, and James Ahrens. Fast Data Parallel Polygon Rendering. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 709–718. ACM, 1993.

[32] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive Ray Tracing for Isosurface Rendering. In *Proceedings of the Conference on Visualization '98*, VIS '98, pages 233–238. IEEE Computer Society Press, 1998.

[33] Tom Peterka, Hongfeng Yu, Robert Ross, and Kwan-Liu Ma. Parallel Volume Rendering on the IBM Blue Gene/P. In *Proceedings of Eurographics Parallel Graphics and Visualization Symposium (EGPGV 2008)*, pages 73–80, April 2008.

[34] Paolo Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. *SIGGRAPH Computer Graphics*, 22(4):51–58, 1988.

[35] Rudrajit Samanta, Thomas Funkhouser, and Kai Li. Parallel Rendering with K-Way Replication. In *Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '01, pages 75–84. IEEE Press, 2001.

[36] Peter Shirley, Michael Ashikhmin, Michael Gleicher, Stephen Marschner, Erik Reinhard, Kelvin Sung, William Thompson, and Peter Willemsen. *Fundamentals of Computer Graphics, 2nd Ed.* A. K. Peters, Ltd., 2005.

[37] Dave Shreiner. *OpenGL Programming Guide 7th Ed.* Addison-Wesley, 2009.

[38] R. Tiwari and T. L. Huntsberger. A Distributed Memory Algorithm for Volume Rendering. In *Scalable High Performance Computing Conference*, pages 247–251, May 1994.

[39] Craig Upson and Michael Keeler. V-buffer: Visible Volume Rendering. In *SIGGRAPH '88: Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, pages 59–64. ACM, 1988.

[40] Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 274–285, 2001.

[41] Hongfeng Yu, Chaoli Wang, Ray W. Grout, Jacqueline H. Chen, and Kwan-Liu Ma. In-Situ Visualization for Large-Scale Combustion Simulations. *IEEE Computer Graphics and Applications*, 30(3):45–57, May/June 2010.